Facultad de Ciencias

APRENDIENDO VECTORES DE PALABRAS A PARTIR DE NORMAS DE ASOCIACIÓN (Learning word embeddings from word association norms)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: **Alejandro Cano Cos**

Director: **Cristina Tîrnăucă**

Septiembre - 2020

Resumen

Hoy en día, los seres humanos utilizamos dispositivos informáticos prácticamente en cada acción que llevamos a cabo; por ello, se hace cada vez más importante facilitar y agilizar la 'comunicación' entre las personas y las máquinas. El campo encargado del estudio de la interacción entre las máquinas y el lenguaje humano es el *Natural Language Processing (NLP)*. El *NLP* trata de conseguir la mejor interpretación posible de la lengua, para lo que se recurre a la modelización del lenguaje, asignando probabilidades a secuencias de palabras mediante algoritmos.

En este trabajo se estudian los *word embeddings*, un tipo de modelado del lenguaje que representa las palabras de un vocabulario como vectores de números reales, lo que permite puntuar la similitud de un par de palabras o mostrar el grado de relación entre ellas. Asimismo, se pueden realizar operaciones entre vectores donde se respetan las analogías entre las palabras.

En particular, se estudiarán algoritmos de generación de estos vectores como ***Word2vec***, ***Node2vec*** y ***Wan2vec***. Estos algoritmos, pese a tener los mismos objetivos, utilizan diferentes fuentes para la obtención de los datos: *Word2vec* utiliza grandes conjuntos de textos escritos, *Node2vec* utiliza grafos y *Wan2vec* utiliza normas de asociación de palabras.

Los objetivos de este trabajo de fin de grado son: el estudio de estos *word embeddings* junto con sus principales algoritmos de generación y la implementación del algoritmo *Wan2vec* con el *dataset Edinburgh Associative Thesaurus (EAT)* introduciendo también el uso de la ontología *Wordnet*.

Palabras clave: *Procesamiento del lenguaje natural, vectores de características, aprendizaje automático, redes neuronales.*

Abstract

Nowadays, we use electronic devices practically in each action we carry out. This is why it is increasingly important to ease and speed up the 'communication' between people and machines. The field in charge of studying the interaction between machines and the human language is called Natural Language Processing (NLP). The NLP seeks to get the best interpretation possible of the languaje. To this end, it resorts to language modelling, assigning probabilities to sequences of words using algorithms.

In this research work, *word embeddings* are investigated. They are a type of language modeling that represent words of a vocabulary with vectors of real numbers. These vectors are able to assign a punctuation to the similarity of a pair of words or to show the relation degree between the words studied. Also, vector operations, where word analogies are observed, can be made.

In particular, vector generation algorithms such as ***Word2vec***, ***Node2vec*** and ***Wan2vec*** will be studied. These algorithms, although they share objectives, use different sources to obtain the data: *Word2vec* uses big sets of written texts, *Node2vec* uses graphs and *Wan2vec* uses word association norms.

The goals of this final degree project are: the study of *word embeddings* along with their main generation algorithms, and the implementation of the *Wan2vec* algorithm with the *Edinburgh Associative Thesaurus(EAT)* dataset, also introducing the use of *Wordnet*.

Keywords: *Natural Language Processing, word embedings, machine learning, neuronal networks.*

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

El Procesamiento de Lenguaje Humano o, en inglés, *Natural Language Processing* (NLP) consiste en la lectura y comprensión del lenguaje escrito o hablado por medio de un ordenador. Es un campo de estudio que se sobrepone a las ciencias de la computación lingüística al estudiar la interacción entre las máquinas y el lenguaje humano, con lo que puede ser considerado un subcampo de la inteligencia artificial.

El uso de dispositivos mecánicos en el desarrollo científico para deshacer las barreras del lenguaje y eliminar las ambigüedades se estudia desde el s.XVII [6]. Sin embargo, se considera que el NLP comienza con Ferdinand de Saussure en los inicios del s.XX con la creación de una nueva teoría del lenguaje. Saussure consideraba el lenguaje natural una estructura de elementos enlazados. Esta forma de ver el lenguaje se conoce como *structuralism* (estructuralismo) y sus ideas han sido fuente de varios trabajos futuros.

Desde entonces se han tratado de desarrollar mecanismos para que las máquinas sean capaces de entender, interpretar y manipular el lenguaje humano de manera eficiente. Estos mecanismos no solo están centrados en entender el lenguaje, sino en comprender además aspectos cognitivos humanos y cómo se organiza la memoria.

Para que un ordenador pueda interpretar el lenguaje natural es necesario modelarlo (o reescribirlo) de alguna manera entendible para un ordenador. Inicialmente, el lenguaje se modelaba con un conjunto de reglas escritas a mano basadas en combinaciones de palabras u otras características. Por ejemplo, se puede construir un sistema para detectar spam mirando si la IP del emisor está en una lista negra o si el correo contiene palabras como *'euros'* y *'has sido seleccionado'*, entre otras. Estas reglas pueden alcanzar una alta precisión si son elaboradas por un experto; no obstante, construir y mantener el sistema de reglas es costoso.

En 1950, Noam Chomsky desarrolló la teoría de las gramáticas generativas y los lenguajes formales [4]. Chomsky demostró que es posible representar un lenguaje potencialmente infinito utilizando gramáticas definidas mediante reglas (o producciones) sobre un conjunto finito de elementos.

A partir de los años ochenta, empezaron a surgir los primeros algoritmos de aprendizaje basados en la estadística[1] que permitían modelar el lenguaje de manera automática a partir de grandes cantidades de datos debidamente etiquetados. Estos modelos asignan una probabilidad a una frase o a cualquier secuencia de palabras. En particular, estos algoritmos han tenido diversas aplicaciones:

- **Etiquetado gramatical**: el modelo generado asigna una probabilidad al etiquetado léxico de una oración. Se pueden utilizar diferentes algoritmos como: n-gramas, árboles de decisión, redes neuronales, etc. [15]. Las aplicaciones del etiquetado gramatical son: compresión del lenguaje, *parsing*, traducción de textos, etc.

- **Recuperación de información**: se asigna una probabilidad a la relevancia de una secuencia de palabras en un documento específico. Se pueden utilizar algoritmos de *clustering*, *k-neighbours*, árboles de decisión, redes neuronales, etc. [3]. Normalmente se emplea en búsquedas sobre grandes documentos de texto.

- **Clasificación de textos**: se asigna la probabilidad de que un texto trate sobre un determinado tema [9]. Se puede utilizar para comprender el lenguaje, clasificar documentos, etc.

- **Similitud y grado de relación entre palabras**: se asigna un valor numérico para medir cómo de parecidas o relacionadas están dos palabras [7].

En la mayoría de estos modelos los ordenadores tienen ciertas dificultades de comprensión, al centrarse mayoritariamente en el significado de cada una de las palabras por separado. Por ejemplo, en modelos de recuperación de la información pueden existir incoherencias debido a la ambigüedad o la polisemia (entre otros factores), como se puede observar en las palabras *banco* o *planta* que tienen dos o más significados.

La gran mayoría de estos algoritmos y reglas están construidos utilizando representaciones simbólicas sobre palabras y no capturan las relaciones entre ellas. De aquí nacen los ***word embeddings***[2], vectores que representan palabras y codifican información permitiendo que el producto escalar sea una medida de similitud entre los términos asociados a cada vector. Cuanto más cercanos sean los vectores entre sí, la similitud entre las palabras que representan es mayor.

Las técnicas utilizadas para generar estos vectores están basadas en la **hipótesis distribucional**: las palabras que aparecen en contextos similares tienden a tener significados parecidos [17]. Algunas de las más conocidas y utilizadas son:

[1] Véase en: https://en.wikipedia.org/wiki/Colorless_green_ideas_sleep_furiously un ejemplo lingüístico atribuido a Chomsky en contra del uso sin criterio de la estadística en el NLP.

[2] A los vectores de palabras se les llama *word embeddings* debido a que los vectores están 'incrustados' en un espacio vectorial. En inglés, *embed* significa incrustar.

- ***LSA*** (*Latent Semantic Analysis*): es un algoritmo que genera un modelo de **recuperación de la información**. Los pesos de los vectores se calculan en torno a la frecuencia de la palabra en un documento [8]. Gracias a la utilización de *word embeddings*, búsquedas como *'large desk'* o *'big table'* generarían un conjunto similar de información relevante.

- ***Word2vec***: introducido por Mikolov et al. [12] en los laboratorios de Google, es un programa que utiliza grandes cantidades de texto para optimizar unos vectores objetivo inicializados aleatoriamente, haciendo uso de redes neuronales.

- ***Glove***: introducido por profesores de la universidad de Stanford [14]. Mezcla LSA y *Word2vec* de manera que hay palabras que tienen más peso sobre otras debido a la frecuencia con la que aparecen. Además, agrupa las palabras de manera no supervisada de forma similar a *Word2vec*.

- ***Node2vec***: introducido por profesores de la universidad de Stanford [5], es un algoritmo que genera caminos aleatorios a partir de un grafo creando así secuencias de palabras (con o sin sentido) utilizadas como entrada para *Word2vec*.

- ***Wan2vec***: desarrollado a partir de *Node2vec*, propone una idea para la creación de estos grafos utilizando **WANs** (*Word Association Norms*, es decir, Normas de Asociación de Palabras), que son agrupaciones de palabras creadas por participantes humanos. Para ello, se comunica una palabra estímulo a los participantes que inmediatamente deben responder con la primera palabra que se les ocurre. Por ejemplo, si se utiliza como palabra estímulo *coche*, los participantes podrían decir: *ruedas*, *volante*, *velocidad*... Cada participante solo responde con una palabra a cada estímulo y todos los estímulos se les presentan a todos los participantes. Una palabra utilizada como estímulo puede tener varias respuestas diferentes, las cuales son almacenadas junto con su frecuencia de aparición [2]. Este algoritmo utiliza las funciones: IF (*Inverse Frequency* o Frecuencia Inversa) o IAS (*Inverse Association Strength* o Fuerza de Asociación Inversa) para asignar pesos a las aristas del grafo.

Word2vec es el algoritmo más popular utilizado para la generación de *word embeddings*. Sin embargo, presenta varios inconvenientes en su implementación sobre diferentes lenguas. En particular, son necesarios conjuntos de textos muy grandes y completos para poder crear un modelo aceptable, lo cual en muchos idiomas no es posible. Además, para implantar *Word2vec* es necesario gran poder de cómputo dado que en muchos casos se trabaja con conjuntos de textos muy grandes y lentos de procesar.

Por ello, el objetivo de este trabajo es el estudio de las alternativas existentes para la generación de *word embeddings* sin necesidad de estos textos y sin utilizar muchos recursos computacionales. En particular, se utilizará el algoritmo *Wan2vec* junto con unas modificaciones que proponemos. El *dataset* empleado

en este trabajo es el ***EAT*** (*Edinburgh Associative Thesaurus*[3]), un conjunto de normas de asociaciones de palabras realizadas por diferentes usuarios. Este dataset contiene alrededor de 20.000 palabras, entre las cuales establecen 500.000 asociaciones de *estímulo-respuesta*. Como objetivo secundario, se ha desarrollado el algoritmo ***Can2vec***. *Can2vec* se presenta como una posible mejora para el algoritmo *Wan2vec* y analiza la adecuación de fortalecerlo con la información proporcionada por una ontología de tipo *Wordnet*[4].

[3]Disponible en: http://vlado.fmf.uni-lj.si/pub/networks/data/dic/eat/Eat.htm

[4]Wordnet es una gran base de datos léxica del ingles creada por la universidad de Princeton. Véase en: https://wordnet.princeton.edu/

Capítulo 2

Fundamentación teórica

2.1. Vectores de características

Las palabras de un vocabulario W de cardinalidad $|W| = n$ se pueden representar fácilmente como vectores en un espacio vectorial de dimensión n. Por ejemplo, en el caso del vocabulario $W = \{casa, computador, \ldots, ordenador, ropa\}$ con $|W| = 1000$, esta representación se podría llevar a cabo de la siguiente manera:

Casa	*Computador*	...	*Ordenador*	*Ropa*
1	0	...	0	0
0	1	...	0	0
⋮	⋮	⋮	⋮	⋮
0	0	...	1	0
0	0	...	0	1

Tabla 2.1: *One-hot encodings*

Cada palabra tiene asignado un vector diferente construido introduciendo para cada palabra un uno en una posición específica (coincidiendo con el índice de la palabra en el vocabulario) y rellenando las demás posiciones con ceros. En este caso particular, cada vector se rellenaría con un uno y 999 ceros. Esta representación recibe el nombre de ***one-hot encoding***.

Los vectores *one-hot* de un vocabulario son muy sencillos de representar; sin embargo, estos vectores no guardan ninguna relación entre sí, mientras que las palabras del vocabulario sí que están relacionadas entre ellas. Por ejemplo, las palabras *'ordenador'* y *'computador'* prácticamente significan lo mismo, pero los vectores *one-hot* de ambas palabras guardan la misma relación entre ellos que con cualquier otro vector de otra palabra distinta. Esto se debe a que el producto escalar entre cualquier par de vectores es cero y la distancia entre todos los vectores es la misma.

Debido a estos inconvenientes, se han propuesto los **vectores de características**. Un vector de características almacena propiedades de una palabra en cada una de sus componentes. Por ejemplo, dado el vocabulario $W = \{Mujer, Hombre, Reina, Rey, Corona, Espagueti\}$ se le asigna a cada palabra del vocabulario un vector de dimensión tres. Las componentes del vector se describen por una característica del conjunto $D = \{Género, Realeza, Comida\}$. En la Tabla 2.2 se muestran a modo de ejemplo los vectores de características de las palabras del vocabulario W.

	Mujer	*Hombre*	*Reina*	*Rey*	*Corona*	*Espagueti*
Género	1	-1	1	-1	0.01	0.05
Realeza	0.03	0.03	1	1	1	0
Comida	0.01	0.01	0.01	0.01	0.01	1

Tabla 2.2: Representación de los vectores caracterizados.

En esta representación se puede observar que los vectores están distribuidos en el espacio de forma diferente a los *one-hot*. Los vectores *Mujer* y *Reina* comparten la componente de género, tomando esta el valor contrario en los vectores *Hombre* y *Rey*. Además, restando *Mujer* a *Reina* u *Hombre* a *Rey* queda un vector parecido a *Corona* ya que se anula el género (queda en valor cero). Asimismo, restando *Mujer* a *Reina* y sumando *Hombre* queda un vector similar a la palabra *Rey*. A esto último se le conoce como **analogía de palabras** y, como se puede apreciar, las operaciones sobre estos vectores están correlacionadas con el significado de las palabras que representan.

Otra operación que se utiliza con estos vectores es el **producto escalar**, cuyo resultado indica intuitivamente la similitud entre dos vectores. Por ejemplo, en la siguiente operación:

$$\begin{pmatrix} 1 \\ 0{,}03 \\ 0{,}01 \end{pmatrix}^T_{Mujer} * \begin{pmatrix} 0{,}01 \\ 1 \\ 0{,}01 \end{pmatrix}_{Corona} = 0{,}0401 \approx 0,$$

el producto escalar del vector *Mujer* con el vector *Corona* da como resultado un numero muy cercano al cero. En este caso las palabras no comparten ninguna característica, es decir, las componentes de ambos vectores no coinciden una a una con valores positivos o negativos suficientemente grandes. Sin embargo, realizando el producto escalar de los vectores *Reina* y *Corona*:

$$\begin{pmatrix} 1 \\ 1 \\ 0{,}01 \end{pmatrix}^T_{Reina} * \begin{pmatrix} 0{,}01 \\ 1 \\ 0{,}01 \end{pmatrix}_{Corona} = 1{,}1001,$$

se obtiene como resultado 1,1001, lo cual indica que ambas palabras comparten ciertas características.

Normalmente las características de los *word embeddings* no se pueden interpretar como en el ejemplo de arriba y suelen tener bastantes más dimensiones

(300 es un ejemplo típico). Sin embargo, aumentar la dimensión puede complicar la visualización de las relaciones entre los vectores, así que para evaluar una representación se suele hacer una proyección a 2D. Por ejemplo, para un vocabulario de tamaño 100k de palabras en inglés con *embeddings* pre-entrenados para cada palabra, se escogen las siguientes palabras: *queen, man, woman, king, computer, calculation, spaghetti, macaroni, crown, ham, meat*. Para pintar estas palabras en 2D se aplica el algoritmo de reducción de dimensionalidad T-SNE [18] a todas ellas y se obtiene la representación que se muestra en la Figura 2.1.

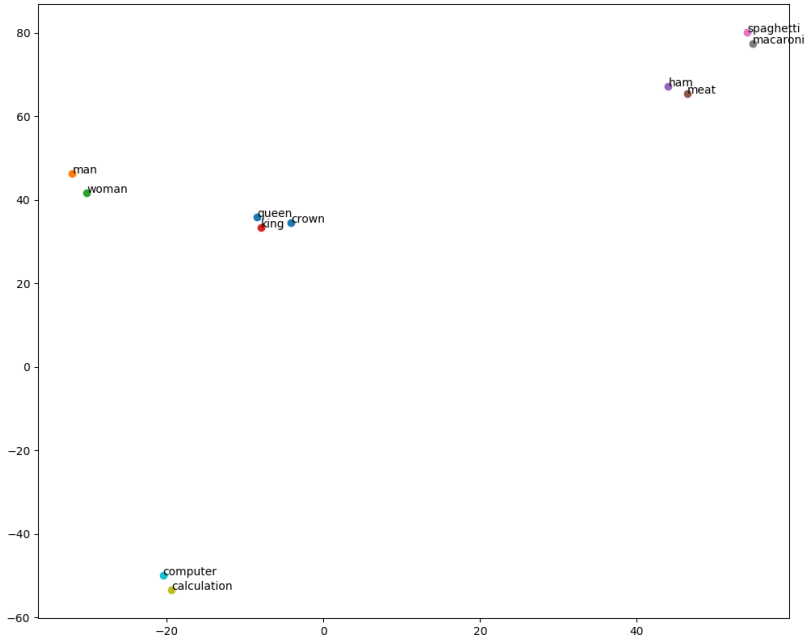


Figura 2.1: Vectores de palabras en 2D.

En la Figura 2.1 se puede apreciar la distribución de las palabras en el plano. Los vectores de palabras que están relacionados como *calculation* y *computer* o *king, queen* y *crown* aparecen agrupados. Incluso dentro de los grupos pueden aparecer subgrupos. Por ejemplo, dentro del grupo de la comida aparecen subgrupos como *ham* y *meet* o *spaghetti* y *macaroni*.

2.2. *Word2vec*

Para la creación de *word embeddings* que representen de manera razonable las relaciones entre palabras en el espacio se necesita un conjunto de datos que contenga las conexiones entre todos los términos. Estas conexiones están determinadas por **pares de palabras**. Un par de palabras construye un enlace entre una palabra **objetivo** o **raíz** y una palabra **contexto** que de algún modo están relacionadas, como por ejemplo: (*agua*, *azul*) o (*hielo*, *frío*).

Para la ejecución de *Word2vec* son necesarios los siguientes elementos:

- Grandes conjuntos de texto escrito no etiquetado, comúnmente denominados **corpus**, de donde se extraen pares de palabras. En el caso de *Word2vec* estos pares se construyen con palabras que están cercanas en el texto. En ocasiones, las palabras que forman los pares no guardan aparentemente una relación semántica entre sí. Por ejemplo, si en el texto a procesar aparece la oración: '*El calvo está en la peluquería*', las palabras *calvo* y *peluquería* podrían formar un par según *Word2vec*, aunque la relación entre ambas no sea coherente. No obstante, es poco común que se forme este tipo de pares y depende, principalmente, de la calidad de los datos de entrada. Por ello, es importante disponer de un corpus grande para así minimizar expresiones poco comunes y/o utilizar textos revisados para evitar incoherencias.

- Tamaño de **ventana**: es el número de palabras contexto seleccionadas de un texto antes y después de una palabra objetivo. Cada palabra contexto forma un par diferente con la palabra objetivo. Todos los pares tienen la misma influencia sobre la palabra objetivo y no se tiene en cuenta la distancia a esta. Un ejemplo de la extracción de pares de palabras de una frase se encuentra en la Figura 2.2[1].

Source Text	Training Samples generated from source text			
I will have orange juice and eggs for breakfast	(will, I)	(will, have)	(will, orange)	
I will have orange juice and eggs for breakfast	(have, I)	(have, will)	(have, orange)	(have, juice)
I will have orange juice and eggs for breakfast	(orange, will)	(orange, have)	(orange, juice)	(orange, and)
I will have orange juice and eggs for breakfast	(juice, have)	(juice, orange)	(juice, and)	(juice, eggs)
I will have orange juice and eggs for breakfast	(and, orange)	(and, juice)	(and, eggs)	(and, for)
I will have orange juice and eggs for breakfast	(eggs, juice)	(eggs, and)	(eggs, for)	(eggs, breakfast)
I will have orange juice and eggs for breakfast	(for, and)	(for, eggs)	(for, breakfast)	

Figura 2.2: Pares extraídos con un tamaño de ventana igual a dos.

[1]Fuente de la imagen: https://towardsdatascience.com/nlp-101-word2vec-skip-gram-and-cbow-93512ee24314

- **Vectores de entrenamiento**: Cada palabra está representada por dos vectores distintos que se inicializan de manera aleatoria, con una dimensión fijada d:

  - Vectores **objetivo**: son los *word embeddings* que buscamos optimizar. Representan en el espacio una palabra de nuestro vocabulario.
  - Vectores **contexto**: son vectores que representan a las palabras cuando actúan de contexto. Se utilizan para optimizar los vectores objetivo. Por ejemplo, en la Figura 2.2, si se optimiza el vector objetivo de la palabra *orange*, se utilizarán vectores contexto de palabras como: *will*, *have*, *juice* y *and*.

Aunque puede parecer un poco lioso utilizar dos representaciones vectoriales diferentes para cada palabra, en la práctica funciona y cada una tendrá su papel.

Al final, el objetivo del algoritmo es maximizar el producto escalar o la similitud entre el vector objetivo de una palabra con los vectores contexto de las palabras que se encuentran en sus pares. Por ejemplo, si se tiene un corpus con las siguientes oraciones:

El zumo de fresa es refrescante
El zumo de kiwi es refrescante

a partir del cual se extraen los pares de palabras, con un tamaño de ventana igual a tres:

$El \rightarrow (El, zumo)$, (El, de), $(El, fresa)$
$zumo \rightarrow (zumo, El)$, $(zumo, de)$, $(zumo, fresa)$, $(zumo, es)$
$de \rightarrow (de, El)$, ..., $(de, refrescante)$
$fresa \rightarrow (fresa, El)$, $(fresa, zumo)$, $(fresa, de)$, $(fresa, es)$, $(fresa, refrescante)$
$es \rightarrow (es, zumo)$. ..., $(es, refrescante)$
$refrescante \rightarrow (refrescante, de)$, $(refrescante, fresa)$, $(refrescante, es)$

$El \rightarrow (El, zumo)$, (El, de), $(El, kiwi)$
$zumo \rightarrow (zumo, El)$, $(zumo, de)$, $(zumo, kiwi)$, $(zumo, es)$
$de \rightarrow (de, El)$, ..., $(de, refrescante)$
$kiwi \rightarrow (kiwi, El)$, $(kiwi, zumo)$, $(kiwi, de)$, $(kiwi, es)$, $(kiwi, refrescante)$
$es \rightarrow (es, zumo)$. ..., $(es, refrescante)$
$refrescante \rightarrow (refrescante, de)$, $(refrescante, kiwi)$, $(refrescante, es)$

se aprecia que los pares con palabras objetivos *fresa* y *kiwi* comparten contextos en las diferentes frases:

$(fresa, El)$ y $(kiwi, El)$
$(fresa, zumo)$ y $(kiwi, zumo)$
$(fresa, de)$ y $(kiwi, de)$
$(fresa, es)$ y $(kiwi, es)$
$(fresa, refrescante)$ y $(kiwi, refrescante)$

y con esta información, los vectores objetivo de *kiwi* y *fresa* van a ser entrenados/optimizados respecto a los mismos vectores contexto, obteniendo así una mayor similitud entre ambos.

2.2.1. Arquitectura

En el artículo *Efficient estimation for word representations in vector space* [10], se presentan dos arquitecturas diferentes para entrenar los vectores de palabras (véase Figura 2.3):

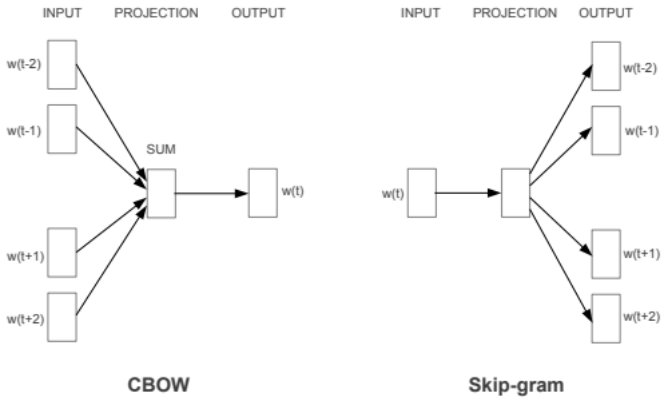


Figura 2.3: Arquitecturas CBoW y Skip-gram. Fuente: [11]

- **Continuous Bag-of-Words**: utiliza n palabras contexto para predecir una palabra objetivo. Predice mejor las relaciones que aparecen en el texto con más frecuencia, lo cual perjudica a palabras que aparecen poco en los mismos contextos.

- **Skip-Gram Model**: utiliza una palabra objetivo para predecir sus palabras contexto, beneficiando así a las palabras que aparecen con menor frecuencia, pero por el contrario, perjudicando a las polisémicas.

En la implementación original de *Word2vec* se emplea la arquitectura *Skip-gram*, utilizando una palabra objetivo como *input*, y una palabra contexto como *output*. Es la arquitectura que utilizaremos en adelante para explicar los demás conceptos relacionados con este modelo.

2.2.2. Función objetivo

El objetivo de *Word2vec* es poder predecir las palabras contexto a partir de una palabra objetivo (en el caso de *Skip-Gram*; al contrario para *CBOW*). Para ello se define la siguiente función a optimizar (maximizar):

$$J'(\theta) = \prod_{t=0}^{|T|} \prod_{-m \leq j \leq m; j \neq 0} p(w_{t+j}|w_t; \theta)$$

donde p es una función que mide la probabilidad de que una palabra objetivo w_t tenga como contexto una palabra w_{t+j} y se desarrollará en la Sección 2.2.3; $|T|$ representa la longitud total de la lista/texto de palabras de donde extraemos los datos y m es el tamaño de la ventana; w_t representa la t-iésima palabra; θ es la matriz de los vectores contexto y objetivo de todas las palabras (que se pretenden optimizar).

Aplicando logaritmos y normalizando J' para que la dimensión del texto no influya en el coste final, obtenemos:

$$J(\theta) = \frac{1}{|T|} \sum_{t=0}^{|T|} \sum_{-m \leq j \leq m; j \neq 0} L(w_{t+j}, w_t; \theta)$$

donde L representa las pérdidas por cada par:

$$L(a, b; \theta) = -\log(p(a|b; \theta))$$

Para minimizar esta función de coste se puede utilizar el método del gradiente descendente, que en cada iteración cambiará las representaciones de los vectores objetivo y contexto (que juntos forman θ) para optimizar la función J y reducir las pérdidas. No se detallará aquí porque no entra dentro de los objetivos de este trabajo.

2.2.3. Función similitud

Para entrenar el modelo se necesita una función de similitud $p(a|b; \theta)$. Esta función calcula la probabilidad de que a sea una palabra contexto de la palabra objetivo b y para ello se utilizan las representaciones de los vectores almacenadas en θ. Se pueden utilizar las siguientes funciones (se omite el parámetro θ para simplificar la notación):

Sotfmax

Utiliza la siguiente función para calcular la probabilidad de que una palabra c sea contexto de una palabra objetivo w:

$$p(c|w) = \frac{\exp(V_c^T * U_w)}{\sum_{c' \in W} \exp(V_{c'}^T * U_w)}$$

donde V_c es el vector contexto de la palabra c, U_w es el vector objetivo de w y W es el conjunto de todas las palabras. Los vectores están extraídos de la variable θ. A continuación, se presenta un ejemplo de uso de la función *softmax*.

Dado un vocabulario $W = \{barco, plaza, ancla, mar\}$, la oración (en forma de lista) $T = [ancla, mar, barco]$ y el tamaño de ventana igual a uno, se generan

los siguientes pares: $\{(ancla, mar), (mar, ancla), (mar, barco), (barco, mar)\}$; dados los vectores objetivo y contexto de dimensión $d = 3$ iniciados aleatoriamente y que juntos forman la matriz θ representada abajo, procedemos a calcular el *softmax* de la palabra objetivo *ancla* con el par $(ancla, mar)$ proveniente de la lista T. La matriz θ tiene el siguente aspecto:

$$\underbrace{\begin{bmatrix} U^T_{barco} \\ U^T_{plaza} \\ U^T_{ancla} \\ U^T_{mar} \\ \hline V^T_{barco} \\ V^T_{plaza} \\ V^T_{ancla} \\ V^T_{mar} \end{bmatrix}}_{\theta} = \underbrace{\begin{bmatrix} 0{,}5 & 0{,}2 & 0{,}3 \\ 0{,}2 & 0{,}3 & 0{,}3 \\ 0{,}1 & 0{,}5 & 0{,}3 \\ 0{,}9 & 0{,}3 & 0{,}1 \\ \hline 0{,}1 & 0{,}4 & 0{,}9 \\ 0{,}2 & 0{,}1 & 0{,}2 \\ 0{,}8 & 0{,}3 & 0{,}5 \\ 0{,}1 & 0{,}3 & 0{,}7 \end{bmatrix}}_{\theta \in \mathbb{R}^{2d|W|}}$$

Primero empezamos calculando el numerador extrayendo el vector contexto de *mar* y el vector objetivo de *ancla* de la matriz θ:

$$\exp\left(V^T_{mar} * U_{ancla}\right) = \exp\left(\begin{bmatrix} 0{,}1 \\ 0{,}3 \\ 0{,}7 \end{bmatrix}^T_{mar} * \begin{bmatrix} 0{,}1 \\ 0{,}5 \\ 0{,}3 \end{bmatrix}_{ancla}\right) = e^{0{,}37} = 1{,}4477$$

Luego, calculamos el sumatorio del denominador:

$$\sum_{c \in W} \exp\left(V^T_c * U_{ancla}\right) = \exp\left(\begin{bmatrix} 0{,}1 \\ 0{,}4 \\ 0{,}9 \end{bmatrix}^T_{barco} * \begin{bmatrix} 0{,}1 \\ 0{,}5 \\ 0{,}3 \end{bmatrix}_{ancla}\right) +$$

$$+ \exp\left(\begin{bmatrix} 0{,}1 \\ 0{,}3 \\ 0{,}7 \end{bmatrix}^T_{mar} * \begin{bmatrix} 0{,}1 \\ 0{,}5 \\ 0{,}3 \end{bmatrix}_{ancla}\right) + \exp\left(\begin{bmatrix} 0{,}8 \\ 0{,}3 \\ 0{,}5 \end{bmatrix}^T_{ancla} * \begin{bmatrix} 0{,}1 \\ 0{,}5 \\ 0{,}3 \end{bmatrix}_{ancla}\right) +$$

$$+ \exp\left(\begin{bmatrix} 0{,}2 \\ 0{,}1 \\ 0{,}2 \end{bmatrix}^T_{plaza} * \begin{bmatrix} 0{,}1 \\ 0{,}5 \\ 0{,}3 \end{bmatrix}_{ancla}\right) = e^{0{,}48} + e^{0{,}38} + e^{0{,}37} + e^{0{,}13} = 5{,}665$$

y se obtiene el resultado:

$$p(mar|ancla) = \tfrac{1{,}4477}{5{,}665} = 0{,}255$$

Ahora que tenemos el valor de $p(mar|ancla)$, se pueden calcular las pérdidas:

$$L(mar, ancla) = -log(0{,}255) = 1{,}364$$

En el entrenamiento del modelo, se propagarán estas pérdidas hacia atrás optimizando los vectores contexto y los vectores objetivo y minimizando así el coste en cada iteración.

Este proceso de entrenamiento mencionado se realizará también con los demás pares: $\{(mar, ancla), (mar, barco), (barco, mar)\}$ de manera ordenada.

El problema de *softmax* es que calcular las pérdidas de un par requiere iterar sobre la longitud del vocabulario entero $|W|$ (para el cálculo del denominador), por lo cual no es muy eficiente para grandes vocabularios.

Hierarchical softmax

Debido a la complejidad en el cálculo de la función *softmax* se introduce *hierarchical softmax*, que utiliza una metodología diferente para evitar recorrer todas las palabras del vocabulario en el cálculo de p empleando un árbol binario donde cada nodo hoja representa una palabra del vocabulario. A todos los nodos internos y finales del árbol se les asigna un vector aleatorio. El número de nodos internos es $|W| - 1$, y si el árbol está balanceado, la altura del árbol es del orden de $\log_2(|W|)$, haciendo que la longitud $L(c)$ del camino desde la raíz hasta una hoja c sea también del orden de $\log_2(|W|)$.

Sean $n_1, \ldots, n_{L(c)-1}, n_{L(c)}$ los nodos que completan el camino desde la raíz hasta el nodo final c, el cual representa la palabra contexto con la que queremos calcular la similitud. Se define la función p como:

$$p(c|w) = \prod_{j=1}^{L(c)-1} \sigma(d_j \cdot V_w^T * V'_{n_j})$$

donde σ es la función sigmoide, V'_{n_j} es el vector correspondiente al nodo interno n_j y V_w es el vector correspondiente a la palabra w. d_j se define de la siguiente manera:

$$d_j = \begin{cases} 1 & \text{si} \quad n_{j+1} \text{ es hijo derecho de } n_j. \\ -1 & \text{en caso contrario.} \end{cases}$$

Nótese que la función p crea para cada palabra w una distribución de probabilidad sobre los nodos hoja, cumpliéndose que:

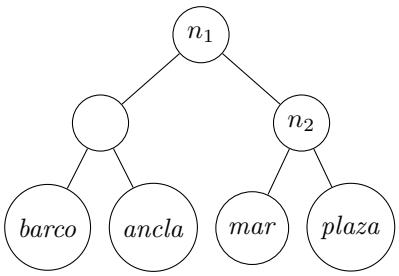
$$\forall w \in W : \sum_{c \in W} p(c|w) = 1$$

Esto se debe a que se empieza desde el nodo raíz con una probabilidad igual a uno, y en el camino al nodo hoja c cada nodo interno genera una distribución de probabilidad entre sus dos hijos, la cual disminuirá en mayor o menor medida la probabilidad acumulada hasta el momento dependiendo del nodo que se escoja y diluirá la probabilidad del nodo no escogido entre los subyacentes de este.

Como la probabilidad de tomar la rama derecha es $\sigma(x)$ y la probabilidad de elegir la rama izquierda es $\sigma(-x)$, se cumple que,

$$\sigma(x) + \sigma(-x) = \frac{1}{1+\exp(-x)} + \frac{1}{1+\exp(x)} = 1$$

A continuación se presenta un ejemplo de *Hierarchical softmax* con el vocabulario: $W = \{barco, plaza, ancla, mar\}$ con el cual se construye el siguiente árbol:



donde los nodos internos y externos ya tienen los pesos asignados. El cálculo de $p(mar|ancla)$ sería de la siguiente manera:

$$p(mar|ancla) = \prod_{j=1}^{L(mar)-1} \sigma(d_j \cdot V_{ancla}^T * V'_{n_j}) =$$

$$\sigma(d_1 \cdot V_{ancla}^T * V'_{n_1}) \cdot \sigma(d_2 \cdot V_{ancla}^T * V'_{n_2}) =$$

$$\sigma(V_{ancla}^T * V'_{n_1}) \cdot \sigma(-V_{ancla}^T * V'_{n_2})$$

Negative sampling

Negative sampling se presenta como una estrategia alternativa a *softmax* y *hicherachical softmax*. El principal inconveniente de ambas funciones está en el coste de la propagación de los errores hacia atrás en la fase de entrenamiento (algo que no llegamos a desarrollar dentro de este trabajo pero que es un tema interesante en sí mismo), debido a que hay una gran cantidad de pesos que modificar en esta propagación. Por eso, en *Negative sampling* se propone utilizar en cada iteración solo un número pequeño prefijado de ejemplos y solamente propagar el error del cálculo sobre los vectores utilizados en vez de actualizar los de toda la red. La gran diferencia con respecto a softmax reside en la capa final. En el caso de *Negative sampling* se emplean nodos de regresión logística (es decir, con la función sigmoide) con el objetivo de clasificar, de la mejor manera posible, los ejemplos positivos y negativos del conjunto de entrenamiento. Los ejemplos positivos se generan de la misma forma que antes; para cada ejemplo positivo (v, w) (con etiqueta uno) se añaden k ejemplos negativos (v, w') (con etiqueta cero) con w' elegido de manera aleatoria. Por lo tanto, es posible que

algunos de estos ejemplos negativos se encuentren realmente en el texto, pero resulta que en la práctica esto no es un inconveniente.

El número k de pares negativos utilizados pasa a ser un parámetro más del algoritmo. Hay varias formas de obtener los ejemplos negativos. Una posibilidad extrema sería hacerlo de manera uniforme (con probabilidad $\frac{1}{|W|}$). Otra opción es que sea acorde a la frecuencia de cada palabra:

$$P(w_i) = \frac{f(w_i)}{\sum_{j=1}^{N} f(w_j)},$$

donde $f(w_i)$ es la frecuencia de aparición de la palabra w_i en el texto y N representa el número de palabras del vocabulario empleado. Tras experimentar con varias opciones, los autores del [2] llegaron a la conclusión de que la que mejor funcionaba era:

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{N} f(w_j)^{3/4}}$$

2.3. *Node2vec*

Node2vec genera textos a través del recorrido de caminos aleatorios sobre los nodos de un grafo. Este algoritmo funciona sobre cualquier tipo de grafo y hace uso de los siguientes componentes para generar los caminos:

- Grafo con vértices y aristas: $G = (V, E)$.
- Una estrategia de recorrido. Hay dos tipos principales de estrategias 'extremas' de recorrido y una suavizada, que es la elegida en nuestras implementaciones. Las estrategias 'extremas' son:
  - **Breadth-first Sampling** (BFS): los nodos explorados son nodos estrictamente adyacentes al nodo fuente. Capta bien las relaciones entre palabras representadas en un mismo contexto, pero mal relaciones generales.
  - **Depth-first Sampling** (DFS): los nodos explorados están a distancias incrementales con el nodo fuente. Al contrario de BFS, capta las relaciones generales entre palabras.

Basado en las dos estrategias mencionadas arriba, la mejor manera de obtener los caminos que empiezan en un mismo nodo es utilizar una estrategia de recorrido híbrida. Se le llama **camino aleatorio** a la secuencia de los n nodos generados mediante la siguiente distribución:

$$P(x|v) = \begin{cases} \frac{\pi_{vx}}{Z} & \text{si} \quad (v,x) \in E \\ 0 & \text{en caso contrario} \end{cases}$$

donde $\frac{\pi_{vx}}{Z}$ es la probabilidad de la transición entre v y x, normalizada con la constante de normalización Z.

Para definir la probabilidad de una transición π_{vx} se usan dos parámetros p y q que guían el camino:

- Parámetro de retorno p: controla la probabilidad de revisitar un nodo en el camino. Poniendo un valor alto disminuye la probabilidad de revisitar un nodo en los próximos dos pasos. Sin embargo, un valor pequeño favorece la repetición de nodos y mantener los caminos en nodos cercanos al nodo raíz.
- Parámetro In-out q: controla la probabilidad de visitar nodos no adyacentes al nodo anterior visitado; un valor pequeño genera caminos más propensos a la exploración de nodos más lejanos.

Para un camino que ha tomado la arista (t, v) y ahora está en v se necesita decidir sobre el próximo paso. Por lo tanto, el algoritmo evalúa la probabilidad de tomar las aristas que cumplen $(v, x) \in E$:

$$\pi_{vx} = \alpha_{p,q}(t, x) w_{v,x}$$

$$\alpha_{p,q}(t, x) = \begin{cases} \frac{1}{p} & si \quad d_{tx} = 0 \\ 1 & si \quad d_{tx} = 1 \\ \frac{1}{q} & si \quad d_{tx} = 2 \end{cases}$$

- $w_{v,x}$ es el peso de la arista (v, x). En caso de no tener pesos asignados, el peso de cada arista toma el valor 1.
- d_{tx} denota el camino más corto entre nodos t y x.

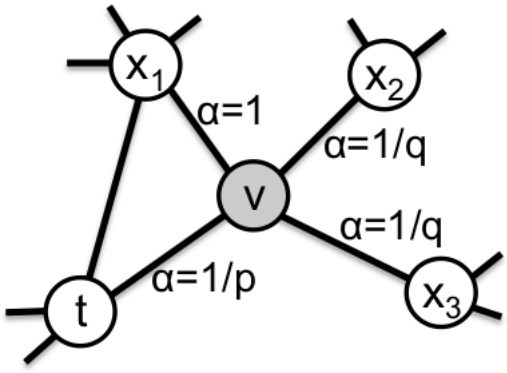


Figura 2.4: Evaluación en nodo v del siguiente paso. Fuente: [5]

En la Figura 2.4 la estrategia de recorrido ha decidido visitar el nodo v tras haber visitado el nodo t y está evaluando las probabilidades de tomar la siguiente arista utilizando la función $\pi_{vx_i} = \alpha_{p,q}(t, x_i) w_{v,x_i}$. Suponiendo que $\forall i$, $w_{v,i} = 1$:

- π_{v,x_1}: $d_{tx_1} = 1$, luego $\alpha_{p,q}(t, x_1) = 1$ y $\pi_{vx_1} = 1$
- π_{v,x_2}: $d_{tx_2} = 2$, luego $\alpha_{p,q}(t, x_2) = \frac{1}{q}$ y $\pi_{vx_2} = \frac{1}{q}$
- π_{v,x_3}: $d_{tx_3} = 2$, luego $\alpha_{p,q}(t, x_3) = \frac{1}{q}$ y $\pi_{vx_3} = \frac{1}{q}$
- $\pi_{v,t}$: $d_{tt} = 0$, luego $\alpha_{p,q}(t, t) = \frac{1}{p}$ y $\pi_{vt} = \frac{1}{p}$

Una vez normalizados estos valores, se tomaría uno de los cuatro nodos siguiendo la distribución de probabilidad.

El algoritmo *Node2vec* en su ejecución itera por todos los nodos de un grafo recibido como *input* y genera un número prefijado de caminos aleatorios con una longitud seleccionada. También es importante escoger unos buenos valores para p y q ya que afectarán a la estrategia de recorrido y al resultado final.

Comparando *Word2vec* y *Node2vec* podría deducirse que las frases extraídas de un corpus y los caminos calculados de un grafo tienen la misma función. Son dos formas diferentes de extraer un tipo de información similar pero de fuentes diferentes. Los caminos de *Node2vec* se pueden recorrer para extraer pares de palabras y entrenar un conjunto de *word embeddings* de la misma forma que en *Word2vec*.

2.4. *Wan2vec*

Wan2vec Utiliza el algoritmo *Node2vec* sobre grafos generados con asociaciones de palabras. La definición formal del grafo es:

- $N = \{v_i | i = 1, ..., n\}$ conjunto de nodos. Cada nodo representa una palabra del vocabulario W y por lo tanto se usará la misma notación pero denota indistintamente una palabra o el nodo del grafo correspondiente a esta palabra.
- $E = \{(v_i, v_j) | v_i, v_j \in V, 1 \leq i, j \leq n\}$ conjunto de aristas.
- $\phi : E \to \mathbb{R}$ función de peso sobre las aristas.

Para la definición de ϕ se proponen diferentes funciones en el artículo que introduce el algoritmo *Wan2vec* [2]:

- **Frecuencia inversa**, $\phi_{IF}(v, w)$: para esto se introduce la **frecuencia** $\phi_F(v, w)$, que cuenta el número de ocurrencias de una respuesta w asociada a un estímulo v. La frecuencia inversa se calcula según la fórmula:

$$\phi_{IF}(v, w) = (\sum_{x \in R_v} \phi_F(v, x)) - \phi_F(v, w)$$

  donde R_v es el conjunto de palabras respuestas al estímulo v

  Por ejemplo, la palabra *banco* utilizada como estímulo obtiene las respuestas $(dinero, 5)$, $(sentado, 3)$ y $(pez, 2)$:

$$\phi_{IF}(banco, dinero) = (5 + 3 + 2) - 5 = 5$$
$$\phi_{IF}(banco, sentado) = (5 + 3 + 2) - 3 = 7$$
$$\phi_{IF}(banco, pez) = (5 + 3 + 2) - 2 = 8$$

- **Fuerza de enlace inversa,** $\phi_{IAS}(v, w)$: para esto se introduce la **fuerza de enlace** $\phi_{AS}(v, w)$, que establece una relación entre la frecuencia y el número de asociaciones para todos los estímulos:

$$\phi_{AS}(v, w) = \frac{\phi_F(v, w)}{\sum_{x \in R_v} \phi_F(v, x)}$$

  Ahora se define la inversa:

$$\phi_{IAS}(v, w) = 1 - \phi_{AS}(v, w)$$

Ya creado el grafo con estos elementos, se puede dar el paso al algoritmo **Node2vec** para generar el conjunto de datos.

2.5. *Can2vec*

Can2vec es el algoritmo desarrollado en este trabajo que utiliza *Node2vec* sobre asociaciones de palabras. Adicionalmente puede usarse con *Wordnet* para ampliar la información que se puede obtener de un *dataset*. La definición formal del grafo G utilizado es:

- $V = \{v_i | i = 1, ..., n\}$, conjunto de nodos que representan las palabras (estímulos y respuestas).
- $E = \{(v_i, v_j) | v_i, v_j \in V, 1 \leq i, j \leq n\}$, conjunto de aristas.
- $\phi : E \to \mathbb{R}$, función de peso sobre las aristas. Esta función tiene el siguiente aspecto:

$$\phi_{FF}(v, w) = \phi_F(v, w) + \phi_F(w, v)$$

Con esta definición de ϕ, podemos usar G como un grafo no dirigido porque se cumple la siguiente condición:

$$\forall w \forall v \ \phi_{FF}(v, w) = \phi_{FF}(w, v)$$

pero tras asignar el peso a todas las aristas y experimentar con esta función, se decidió normalizar el peso de cada una de ellas respecto a cada nodo. Esto hace que una arista tenga un peso diferente según la influencia que tenga en el nodo. La normalización se realiza con la siguiente función:

$$\phi_{FFN}(v, w) = \frac{\phi_{FF}(v,w)}{\sum_{z \in V} \phi_{FF}(v,z)}$$

donde v es el nodo que se quiere normalizar. La función ϕ_{FFN} permite que las aristas que contengan un nodo asociado a muchos nodos no tengan la misma probabilidad de ser atravesadas desde sus diferentes extremos.

Recordemos que el algoritmo utilizado para la generación de pares de palabras es *Node2vec*. Este genera caminos utilizando una distribución de probabilidad discreta en la selección de cada nodo, asignando una probabilidad de expandir a un nodo directamente proporcional al peso de su arista. Además, los pesos de las aristas tienen que representar cómo de fuerte es un enlace entre una palabra y su contexto, sin diferenciar ninguna direccionalidad debido a que en un texto sin etiquetar tampoco se diferencia. Por todo esto, se ha definido ϕ como la suma de las frecuencias de los nodos que componen la arista. No obstante, al normalizar los pesos quedará un grafo dirigido.

Con el grafo definido, se puede llevar a cabo la fase de ampliación de información. Para entender esta fase es necesario comprender *Word2vec*. Este algoritmo basa su funcionamiento en la hipótesis distribucional (palabras de contextos similares tienen significados parecidos) por la cual se puede extraer del texto información de una forma determinada. Los grafos generados a partir de *WANs* también tienen el objetivo de ser explorados bajo esa hipótesis debido a que el contenido que genera será la entrada de *Word2vec* para entrenar un modelo. Por ello, se propone reforzar el grafo siguiendo la otra faceta de la hipótesis: palabras que tienen significados parecidos tienen contextos similares. Para reforzarlo, se va a utilizar la ontología *Wordnet*, la cual contiene mucha información sobre las palabras y sus relaciones. Dentro de las funciones de wordnet se introduce la función similitud de *wup* [13], definida sobre *Wordnet* como:

$$wup\_sim(w_1, w_2) = 2 * \frac{depth(lcs(w1, w2))}{depth(w1) + depth(w2)},$$

donde w_1 y w_2 son dos palabras, $depth(w)$ es la profundidad de la palabra w en el árbol de *Wordnet* y $lcs(w_1, w_2)$ es el *least common subsummer* o minimo común ancestro entre las palabras w_1 y w_2.

En particular, para cada nodo se pueden calcular sus sinónimos y crear nuevas aristas para crear enlaces, siendo normalizado su peso a la vez que las demás aristas. Para la implementación de la creación de aristas se pueden utilizar diferentes variantes:

- Transferencia completa de aristas: cada palabra pasa todas sus aristas con sus pesos iguales a todos sus sinónimos.
- Transferencia con *treshold*: se crea la arista dependiendo de la relación entre el sinónimo y el nuevo contexto. Los sinónimos no tienen porque estar relacionados con todos los contextos de su sinónimo. Por ello se añade una medida de la similitud entre ambos lados de la arista. Si la similitud supera el *treshold* entonces se añade. Podemos utilizar la similitud *wup* entre el contexto y el sinónimo para comparar con el *treshold*.
- Transferencia completa y normalización con similitud de *wup*. Se transifieren todas las aristas a los sinónimos, pero luego se multiplican todas las

aristas del grafo por la similitud de *wup*.

En caso de añadir una arista a un nodo en la que ya exista, se le asignará la que mayor peso tenga. Tras haber modificado el grafo con los enlaces correspondientes, pasarán a normalizarse todas las aristas del grafo respecto a cada nodo.

2.6. Validación de un modelo

Al no existir soluciones que funcionen bien en todas las aplicaciones, es recomendable testear la calidad de los *word embeddings* de distintas maneras. Antes de dar paso a la descripción de las técnicas utilizadas, se definen los siguientes coeficientes de correlación utilizados en la gran mayoría de las pruebas [16]:

- Coeficiente de *Spearman*: mide la correlación entre dos variables independientes. El inconveniente que presenta esta medida es que no es independiente de la escala de las variables; por lo tanto, en variables con rangos de valores más pequeños se obtendrán correlaciones más grandes debido a que la distancia entre dos elementos es menor. La ecuación utilizada es:

$$\rho = 1 - \frac{6 \sum D^2}{N(N^2-1)}$$

  donde N representa el número de elementos y $\sum D^2$ la suma al cuadrado de las diferencias entre todos los elementos de los dos conjuntos de datos a comparar.

- Coeficiente de *Pearson*: al igual que *Spearman*, mide la correlación entre dos variables, con la diferencia de que *Pearson* es independiente de la escala de las variables. Para su obtención se emplea la siguiente ecuación:

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

  donde $\bar{x}$, $\bar{y}$ son las medias aritméticas de las series $(x_i)_i$ y $(y_i)_i$, respectivamente.

Con estos coeficientes definidos, se presentan los dos tipos de evaluación:

- **Evaluación intrínseca**: prueba el modelo en una subtarea. Es más rápido de ejecutar, aunque en ocasiones optimiza únicamente problemas locales. Algunas maneras de evaluar el modelo son:

  - Comprobar los resultados obtenidos al utilizar **analogías** de palabras: hombre para mujer es lo mismo que reina para el rey, $V_{reina} \sim V_{rey} - V_{hombre} + V_{mujer}$. Para comparar la similitud entre el vector reina y el vector obtenido combinando los vectores correspondientes a rey, hombre y mujer se utiliza la **similitud del coseno**:

$$cos\_sim(v, x) = \frac{v^T * x}{||v|| \cdot ||x||}$$

donde v y x son los vectores que se quieren comparar. Para hacer un test con una analogía de palabras, lo ideal es que el vector calculado $x = V_{rey} - V_{hombre} + V_{mujer}$ maximize la función de la similitud con el vector reina:

$$argmax_v \ cos\_sim(x, v) = V_{reina}$$

Además, el coseno de los vectores de características también se puede utilizar para calcular el grado de similitud de dos palabras cualesquiera, no solamente para analogías.

- Visualizar los vectores en un plano 2-D: se pueden proyectar los *embeddings* y ver de qué formas se agrupan.
- Comparar conjuntos de distancias/similitud entre vectores de un modelo con distancias obtenidas por juicio humano. Para medir el error entre ambas muestras se utilizan los coeficientes de correlación definidos previamente. Hay varios *benchmarks* creados para evaluar modelos como: BLESS[2], SIM-LEX[3], etc.

- **Evaluación extrínseca**: Evalúa el modelo en una tarea real. Esta evaluación requiere más tiempo y es más costosa que la intrínseca.

[2]Disponible en: https://github.com/alexanderpanchenko/simeval/blob/master/datasets/bless.csv
[3]Disponible en: https://fh295.github.io/simlex.html

Capítulo 3

Construcción del software

En este capítulo se explicará cómo se ha desarrollado el software de generación de *word embeddings*. En primer lugar, se expondrán los requisitos, funcionales y no funcionales, que debe cumplir el software. Posteriormente se procederá a explicar la metodología llevada a cabo para la implementación del programa. Además, se explicará cómo se ha llevado a cabo el diseño y el desarrollo del código y cómo posteriormente se han realizado diversas pruebas y ensayos que determinan que el sistema funciona correctamente.

3.1. Requisitos

Los requisitos que ha de cumplir un software pueden ser funcionales o no funcionales, tal y como se explica en esta sección.

3.1.1. Requisitos funcionales

Los requisitos funcionales de un sistema describen cualquier actividad realizada por el mismo, es decir, los servicios y funciones particulares de un software bajo determinadas condiciones[1]. A continuación se exponen los requisitos funcionales del sistema estudiado:

1. El software debe ser capaz de generar *word embeddings* a través de un grafo, utilizando el algoritmo *Node2vec*.

2. El software debe de soportar la creación de vectores partiendo de grafos dirigidos y no dirigidos.

3. El software debe ser capaz de leer un fichero formateado con las *WANs* y de transformarlo en un grafo.

[1]Una descripción de estos requisitos puede encontrarse en la siguiente página: http://www.pmoinformatica.com/2017/02/requerimientos-funcionales-ejemplos.html

4. El software debe ser capaz de asignar pesos a las aristas de un grafo utilizando la función de *Frecuencia inversa*.

5. El software debe ser capaz de asignar pesos a las aristas de un grafo utilizando la función de *Fuerza de asociación inversa*.

6. El software debe ser capaz de utilizar el algoritmo *Can2vec* y de asignar pesos a las aristas de un grafo utilizando la ontología *Wordnet*.

3.1.2. Requisitos no funcionales

Los requisitos no funcionales del sistema no se refieren a las funciones específicas realizadas por el software, sino a las propiedades del sistema[2]. Se exponen a continuación:

1. Requisito de **rendimiento**: El software debe ser capaz de generar un modelo en un tiempo no superior a 1h.

2. Requisito de **mantenibilidad**: El software debe ser modular para poder minimizar el impacto de la modificación de cualquiera de los elementos que actúan en la generación del grafo. Además, debe estar bien documentado para que se pueda reproducir y modificar por investigadores interesados en el tema.

3. Requisito de **compatibilidad**: El software debe ser compatible con la versión de `Python` 2.7.17 para el uso de librerías desarrolladas en esta versión.

3.2. Metodología

Para llevar a cabo el desarrollo del software se ha utilizado una metodología **incremental**, lo que ha dividido el proyecto en dos etapas:

- Etapa de **iniciación**: abarca la familiarización con el algoritmo *Node2vec* para grafos dirigidos y no dirigidos. El código de *Node2vec* es público y se encuentra publicado en *Github* por la Universidad de Stanford[3], por lo que simplemente se ha tomado como base. Por lo tanto, esta etapa engloba los requisitos funcionales 1 y 2.

- Etapa de **iteración**: esta etapa está dedicada a la implementación de las funciones propias del algoritmo *Wan2vec* para la creación del grafo utilizado en *Node2vec*. Se ha seguido un procedimiento iterativo gracias al cual se han conseguido los siguientes objetivos:

[2]Una explicación de los requisitos no funcionales se encuentra en: https://medium.com/@requeridosblog/requerimientos-funcionales-y-no-funcionales-ejemplos-y-tips-aa31cb59b22a

[3]Código original de *Node2vec*: https://github.com/aditya-grover/node2vec

1. Desarrollo de una función para la lectura de un fichero formateado con las *WANs*, lo que incluye el requisito funcional 3.
2. Desarrollo de las funciones de *Frecuencia inversa* y *Fuerza de asociación inversa* para la asignación de los pesos de las aristas del grafo. Con lo que se abarcan los requisitos funcionales 4 y 5.
3. Desarrollo del algoritmo *Can2vec* junto con la obtención de pesos de las aristas con *Wordnet* lo que engloba al requisito funcional 6. Esto se ha implementado sin la introducción de nuevo vocabulario en el grafo, para cumplir con el requisito no funcional 1.

3.3. Implementación

Para llevar a cabo la implementación del algoritmo *Wan2vec* y posteriormente *Can2vec*, se han desarrollado y utilizado diferentes módulos. Para ello ha sido necesario recurrir a recursos externos que se exponen en los siguientes puntos:

- Código original del algoritmo *Node2vec*[4] desarrollado por la universidad de Stanford para generar el *dataset* utilizado.
- Dataset *EAT* (*Edinburgh Associative Thesaurus*).
- Librerías `Python 2`: `Networkx`[5], `Gensim`[6], `Sklearn`[7], `Numpy`[8], `Random`[9] y `NLTK`[10]. Esto cumple con el requisito no funcional 3.

3.3.1. Diseño y desarrollo

Se ha programado el algoritmo *Wan2vec* haciendo uso del lenguaje de programación `Python`. El desarrollo del programa se puede dividir en dos etapas:

- **Generación del *dataset*** de entrenamiento: para la generación de datos se ha decidido utilizar las normas de asociación del dataset *EAT*. El dataset *EAT* recoge todas las normas de palabras estudiadas en un fichero de texto, con la siguiente estructura:

  $$< PalabraEstímulo > < PalabraRespuesta > < FrecuenciaEnlace >$$

  Desde `Python` se puede construir un grafo que lee este fichero utilizando la librería `Networkx`. Esta librería proporciona los métodos `read_pajek(path)`

[4] https://github.com/aditya-grover/node2vec
[5] https://networkx.github.io/
[6] https://pypi.org/project/gensim/
[7] https://scikit-learn.org/stable/
[8] https://numpy.org/
[9] https://docs.python.org/3/library/random.html
[10] https://www.nltk.org/

o `read_edgelist(path)` que se diferencian en el archivo que leen. Mientras que `read_pajek` lee un fichero con una cabecera donde aparecen etiquetas para los nodos, `read_edgelist` lee el grafo directamente.

Se ha optado por utilzar el método `read_pajek(path)` y tras la lectura del fichero se obtiene un grafo con las conexiones entre las palabras, utilizando la frecuencia de asociación como peso de las aristas. El algoritmo *Wan2vec* utiliza las siguientes funciones para asignar un peso a las aristas:

- *Frecuencia inversa*.
- *Fuerza de asociación inversa*.

El principal inconveniente de la utilización de ambas funciones es que implica que palabras que presenten una relación más estrecha tengan un menor peso en la arista, al igual que las palabras que presentan una relación pequeña obtengan un peso más grande. Esta forma de crear el grafo llevada a cabo en *Wan2vec* no concuerda con el posterior uso del grafo por parte de *Node2vec*. En la creación de caminos en *Node2vec* se utiliza una distribución de probabilidad discreta para escoger qué nodo expandir en cada momento. Asimismo, cada nodo tiene asignada una probabilidad directamente proporcional al peso de la arista, lo cual hace pensar que sea más razonable utilizar la *Frecuencia de asociación* como peso entre las aristas para dar lugar a la creación de secuencias de palabras con mayor frecuencia de emparejamiento entre ellas.

Por lo tanto, en este trabajo de fin de grado se exploran los resultados obtenidos utilizando las frecuencias, modificaciones en el grafo y se proponen otras funciones utilizando la ontología *Wordnet*. Todo ello está recogido dentro de la explicación del algoritmo *Can2vec* en la Sección 2.5.

Construido el grafo con los pesos asociados a cada arista, se puede incrustar el código original del algoritmo *Node2vec* desarrollado por la Universidad de Stanford, que consta de dos ficheros `Python`:

- **Node2vec.py**: contiene la clase Graph, la cual está compuesta por varios métodos:
  - `Node2vec_walk`: simula un camino aleatorio desde un nodo inicial.
  - `simulate_walks`: realiza caminos aleatorios para cada nodo.
  - `preprocess_transition_probs`: preprocesa las probabilidades de las aristas.
  - `get_alias_edge`: obtiene el alias de una arista.
  - `alias_setup`: calcula listas útiles para ejemplos no uniformes de distribuciones discretas.
  - `alias_draw`: dibuja ejemplos de distribuciones discretas no uniformes. En otras palabras, genera un número aleatorio dada una distribución de probabilidad para decidir qué nodo visitar.

- **main.py**: es el fichero main que contiene los siguientes métodos:
  - `parse_graph`: lee los argumentos del programa.
  - `read_graph`: lee grafo de txt.
  - `learn_embeddings`: genera los *word embeddings*.

- **Entrenamiento del modelo**:

  Para el entrenamiento y generación del modelo se ha utilizado la librería `gensim.models.Word2vec` de `Python`[11], con la cual es posible generar vectores de palabras llamando al método *Word2vec* utilizando los siguientes argumentos:

  - `Oraciones (iterable de iterables, opcional)` – puede ser una lista de listas de símbolos o conjuntos de frases.
  - `Tamaño de los vectores (int, opcional)` – Dimensiones de los vectores de palabras.
  - `Tamaño de la ventana (int, opcional)` – distancia máxima entre la palabra actual y la palabra contexto.
  - `min_count (int, opcional)` – ignora todas las palabras con una frecuencia total inferior a esta.
  - `workers (int, opcional)` – número de threads utilizados para entrenar el modelo.
  - `sg (0, 1, opcional)` – Algoritmo de entrenamiento: 1 para skip-gram, sino CBOW.
  - `hs (0, 1, opcional)` – Si es 1 se utilizará *Hierarchical softmax* para el entrenamiento del modelo. Si es 0 se usará *Negative sampling*.

  Tras haber entrenado los vectores de características, se utiliza el método *save_Word2vec_format* para guardarlos en un fichero de texto. La implementación descrita en esta seción se puede encontrar en *GitHub*[12].

[11] Documentación de la liberia disponible en: https://radimrehurek.com/gensim/models/word2vec.html
[12] Código disponible en: https://gitlab.com/canoo/tfg_wordembeddings/-/tree/master

Capítulo 4

Pruebas y resultados

En este capítulo se analizarán las pruebas realizadas y los resultados obtenidos con el algoritmo *Wan2vec* y el *dataset EAT*. También se compararán estos resultados con los obtenidos con *Wordnet*. Debido a que *Wan2vec* es un algoritmo no determinista, produce salidas diferentes para los mismos parámetros y no es posible hacer pruebas de su implementación de manera predefinida. Por lo tanto, para poner a prueba el software se ha comprobado la validez y calidad de los *embeddings* generados mediante las diferentes alternativas.

4.1. Pruebas

En esta sección se describirá el diseño de las pruebas realizadas a los modelos.

4.1.1. Proyección de embeddings a un plano en 2D

Se proyectan los *word embeddings* a un plano en 2D con el algoritmo T-SNE. En esta representación aparecerán agrupadas en distintas regiones del plano las palabras que presenten un alto grado de similitud. Se representarán 4 grupos de palabras con diferentes colores asignados a cada uno de ellos:

- Vehículos (azul): *bike*, *bus*, *train*, *car*.
- Animales (rojo): *mouse*, *dog*, *rabbit*, *cat*.
- Ropa (morado): *sweater*, *shirt*, *dress*, *shoes*.
- Comida (verde): *ham*, *maccaroni*, *bread*, *rice*.

La evaluación de este tipo de prueba es bastante subjetiva ya que no hay ningún baremo para medir la calidad de la representación. No obstante, se lleva a cabo porque resulta intuitivo y es sencillo comprobar si las agrupaciones son erróneas.

4.1.2. Analogías de palabras

También se ha creado un test con las siguientes analogías para evaluar los diferentes modelos:

- $V'_{queen} = V_{king} - V_{man} + V_{woman}$
- $V'_{sculptor} = V_{chef} - V_{food} + V_{stone}$
- $V'_{flower} = V_{tree} - V_{leaf} + V_{petal}$
- $V'_{kitten} = V_{puppy} - V_{dog} + V_{cat}$

De cada modelo se obtendrán las tres palabras más similares al vector calculado a partir de la analogía de palabras. De esta manera es posible evaluar el resultado y sacar conclusiones.

Además, para hacer un test más completo sobre analogías se ha utilizado el fichero[1] con aproximadamente $20k$ analogías de palabras creadas y utilizadas por *Google* para evaluar sus modelos. Este test nos dará como resultado el número de aciertos dividido entre el número de analogías.

4.1.3. Similitud entre palabras

Para hacer más tests sobre los *embeddings*, se ha utilizado el *dataset Wordsim-353* [1], el cual está formado por pares de palabras cuya similitud está definida por expertos. Por ejemplo, a las palabras *love* y *sex* se les asigna una puntuación de 6,77/10. A partir de las asignaciones realizadas se obtiene un conjunto de datos o muestra. Esta muestra se utilizará para el cálculo de correlación con el conjunto de datos generados por el cálculo de la similitud de los pares del *dataset* con unos *embeddings*.

4.1.4. Pruebas al software

Adicionalmente, se ha diseñado una prueba para la comprobación del software implementado. No es una prueba con la que se pueda garantizar que el programa funciona correctamente pero sí puede ser de gran utilidad en el caso de que los resultados obtenidos sean incorrectos.

Con el objetivo de llevar a cabo esta prueba, se ha programado un test para calcular la correlación de Pearson sobre dos muestras que contienen los datos de la similitud entre diferentes palabras. La primera muestra está generada a partir de los *word embeddings* originales de *Wan2vec*; mientras que la segunda muestra se crea a partir de los vectores entrenados con el programa desarrollado en este trabajo, utilizando la *fuerza de asociación inversa*.

[1] Recurso disponible en: https://github.com/nicholas-leonard/word2vec/blob/master/questions-words.txt

Para esta prueba se han generado aleatoriamente 20000 pares de palabras y se ha calculado la similitud entre ellos con los diferentes *embeddings*. La correlación obtenida entre las muestras es de 0.8453, lo cual indica que aproximadamente el 85 % de la información del *EAT* se ha *incrustado* de una forma parecida en ambos conjuntos, dependiendo un 15 % de la aleatoriedad de los caminos escogidos por *Node2vec*, lo cual parece un resultado bastante razonable.

Por otro lado, si se hubiera obtenido una correlación menor que 70 % podría ser un indicativo de que los vectores generados por el programa desarrollado presentan mayor aleatoriedad que los obtenidos con *Wan2vec*, que se ha utilizado un *dataset* diferente, o que el algoritmo está mal programado.

4.2. Resultados

En esta sección se detallarán los resultados obtenidos a partir de las pruebas realizadas.

4.2.1. Proyección de *embeddings* a un plano en 2D

Un ejemplo de la proyección a $2D$ realizada con los *embeddings* de *Wan2vec* se presenta en la Figura 4.1.

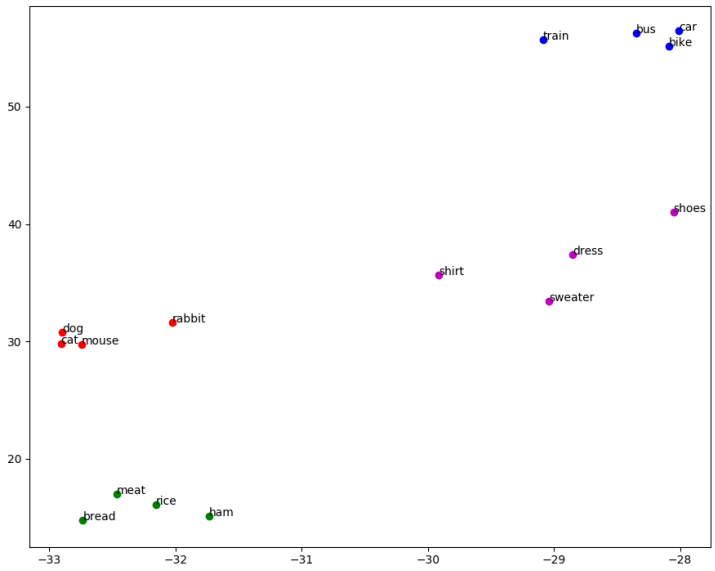


Figura 4.1: Proyección en 2D

En la Figura 4.1 puede apreciarse que los datos están distribuidos de una forma correcta. Se ha comprobado que las proyecciones generadas por los demás modelos son capaces de agrupar las palabras de una forma similar aunque no se adjuntarán las imágenes de los demás modelos porque no se considera una información relevante. Concluyendo, a pesar de que no hay una manera de medir cómo de bien distribuidos están los puntos en el plano, se ha alcanzado el resultado esperado en esta prueba.

4.2.2. Analogías de palabras

La analogías se han calculado con los vectores de *Wan2vec*, con los vectores frecuencia obtenidos en el programa realizado en este trabajo y con vectores preentrenados por *Google*[2] utilizando *Word2vec*. Así, se reportan las tres primeras palabras más parecidas que maximizan la función similitud para cada modelo. Estos datos se presentan en la Tabla 4.1.

	V'_{queen}	$V'_{sculptor}$	V'_{flower}	V'_{kitten}
Wan2Vec ϕ_{IAS}	*princess* *pamela* *maiden*	*finger-orint* *marble* *appleton*	*sepals* *roses* *cream*	*feline* *purr* ***kitten***
Wan2vec ϕ_F	***queen*** *kong* *crown*	*paving* *mason* *rolling*	*piver* *gardener* *tust*	*enigma* *todd* ***kittens***
Can2vec	*monarch* *kong* ***the_queen***	*paving* *onyx* *rolling*	*sepals* ***flowers*** *bouquet*	***kitten*** *pussy* *kittens*
Can2vec + wup treshold	***the_queen*** *reigning* *monarch*	*promenade* *saunter* *amble*	*sepals* ***flowers*** *bouquet*	***kitten*** *pussy* *kittens*
Word2vec (Google)	***queen*** *monarch* *princess*	*marble* ***sculptor*** *granite*	*trees* *rosebush* ***flowers***	***kitten*** *pup* *kittens*

Tabla 4.1: Resultados de analogías de palabras

En la Tabla 4.1 aparecen las palabras obtenidas para cada modelo. Se aprecia que los vectores entrenados con *Can2vec* logran unos resultados similares a los obtenidos con los vectores de *Google*. Sin embargo, los vectores de *Wan2vec* han presentado un funcionamiento notablemente peor, aunque algunas palabras guardan una estrecha relación con la palabra a predecir, como por ejemplo los términos *princess* y *queen*.

Tras este test de analogías, se pasa el test de *Google* con más analogías para una evaluación más completa de los *embeddings*. El porcentaje de acierto de los

[2]Vectores disponibles en: https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit

modelos se muestra en las Tablas 4.2 donde los embeddings han sido entrenados con diferentes dimensiones.

Metaparámetros	d	*Wan2vec* ϕ_{IAS}	*Wan2vec* ϕ_F	*Can2vec*	*Can2vec + wt*
$p = 1,\ q = 1$	120	**0.101**	0.051	0.093	0.092
$p = 1,\ q = 0{,}5$	120	0.095	0.0388	0.0972	0.096
$p = 1,\ q = 2$	120	0.086	**0.0445**	**0.101**	**0.099**
$p = 1,\ q = 1$	300	**0.087**	0.045	**0.0853**	0.082
$p = 1,\ q = 0{,}5$	300	0.08	0.0393	0.0802	**0.089**
$p = 1,\ q = 2$	300	0.086	**0.045**	0.0826	0.089

Tabla 4.2: Resultados del test de analogías de *Google*

Se puede apreciar en general que las puntuaciones son bastante bajas comparadas con los vectores de *Word2vec* pre-entrenados por *Google* que alcanzan una puntuación de 0.65. Esto puede ser debido a las siguientes razones:

- Falta de información en *EAT*: las analogías presentadas en el test son muchas y muy variadas, lo que puede hacer que tal vez hubiera sido necesario tener más participantes en la creación del *dataset* para poder tener un conjunto de datos que capte mejor todas las relaciones.
- Coherencia del grafo: pasar de obtener la información de textos (como hace *Word2vec*) a obtenerla de un grafo es bastante diferente. Es posible que al ser un grafo de grandes dimensiones y no haber una formalización clara del significado de las palabras no sea posible utilizar *Node2vec* para este tipo de tareas o al menos con este tipo de grafos.

4.2.3. Similitud entre palabras

Los resultados del test *Wordsim-353* se presentan en la tabla 4.3.

Metaparámetros	d	*Wan2vec* ϕ_{IAS}	*Wan2vec* ϕ_F	*Can2vec*	*Can2vec + wp*
$p = 1,\ q = 1$	120	**0.71**	**0.258**	**0.679**	0.676
$p = 1,\ q = 0{,}5$	120	0.65	0.123	0.683	**0.686**
$p = 1,\ q = 2$	120	0.6	0.27	0.674	0.685
$p = 1,\ q = 1$	300	**0.66**	**0.25**	0.66	0.6644
$p = 1,\ q = 0{,}5$	300	0.64	0.048	0.665	**0.6759**
$p = 1,\ q = 2$	300	0.59	0.16	**0.678**	0.6575

Tabla 4.3: Resultados para el test *WordSim-353*

Como se aprecia en los resultados, las funciones inversas de *Wan2vec* obtienen unos resultados muy similares a los de *Can2vec* aunque se hayan generado de manera diferente.

El *dataset* de *Google* entrenado con *Word2vec* obtiene en esta prueba una puntuación de 0,55, por lo que los grafos de WANs parecen captar mejor la similitud entre las palabras que utilizando *Word2vec*. Esto puede ser debido a que los vectores se entrenan siempre respecto a palabras cercanas en el grafo, a diferencia de *Word2vec*, que no tiene un 'filtro' para las palabras que no son interesantes para el entrenamiento del algoritmo.

Capítulo 5

Conclusiones

Los algoritmos de generación de *word embeddings* mostrados en este documento son una manera interesante de crear y optimizar vectores de palabras. En primer lugar, se asigna una definición aleatoria a todas las palabras; posteriormente, se 'modifica' esta caracterización agregando datos externos que guían el proceso de aprendizaje de vectores en función de los vectores de las palabras que tienen en su proximidad. Es muy interesante como, a partir de datos no etiquetados (texto plano), se pueden emplear técnicas de aprendizaje supervisado para obtener representaciones de palabras sorprendentemente buenas.

Asimismo, estos vectores permiten ser reentrenados varias veces con diferentes vocabularios o *datasets*. Esto supone una gran ventaja, pues es posible focalizar el modelo en alguna tarea y mantenerlo actualizado debido a que a medida que el tiempo avanza en la sociedad, las interacciones entre las palabras van cambiando y aparecen otras nuevas. Por ejemplo, actualmente la RAE está estudiando integrar la palabra *coronavirus* en el diccionario[1] y probablemente las palabras *virus*, *miedo*, *crisis* y *confinamiento* han establecido nuevas relaciones en nuestra cabeza, debido a los tiempos en los que vivimos.

Respecto a los resultados obtenidos, es probable que para modelar el inglés se requieran más tipos de información o de mejor calidad. Se debe tener en cuenta que la dificultad que existe en conseguir formalizar el lenguaje natural es muy grande. En *Wan2vec* se intenta primeramente conectar las palabras de un grafo de donde se extrae la información, a diferencia de *Word2vec* que directamente extrae la información de grandes conjuntos de texto no etiquetado. Pero, ¿es realmente posible conseguir representar las interacciones de las palabras en un grafo para alcanzar mejores resultados que los obtenidos utilizando grandes corpus? De momento, las propuestas existentes no parecen indicar que sea el caso.

Una vez finalizado el trabajo, se considera que los objetivos establecidos al inicio del mismo se han cumplido: se ha estudiado a fondo el funcionamiento de los *word embeddings* junto con sus algoritmos de generación y además se

[1] Noticia de la RAE: https://www.rae.es/noticias/crisis-del-covid-19-sobre-la-escritura-de-coronavirus#:~:text=La

han planteado y estudiado nuevas propuestas, incluyendo el uso de la ontología *Wordnet*.

Bibliografía

[1] Eneko Agirre, Enrique Alfonseca, Keith Hall, Jana Strakova, Marius Pasca, and Aitor Soroa. A study on similarity and relatedness using distributional and wordnet-based approaches. pages 19–27, 01 2009.

[2] Gemma Bel-Enguix, Helena Gomez Adorno, Jorge Reyes-Magaña, and Gerardo Sierra. Wan2vec: Embeddings learned on word association norms. *Semantic Web*, 10:991–1006, 10 2019.

[3] P. K. Bhatia, Tanya Mathur, and T. Gupta. Survey paper on information retrieval algorithms and personalized information retrieval concept. *International Journal of Computer Applications*, 66:14–18, 2013.

[4] Igor A. Bolshakov and Alexander Gelbukh. *Computational linguistics: Models, Resources, Applications*. 2004.

[5] Aditya Grover and Jure Leskovec. Node2vec: Scalable feature learning for networks. *CoRR*, abs/1607.00653, 2016.

[6] W. J. Hutchins. *Machine Translation, past, present and future*. John Wiley & Sons, Inc, 1986.

[7] Peter Kolb. Experiments on the difference between semantic similarity and relatedness. *NODALIDA 2009 Conference Proceedings*, 4, 01 2009.

[8] Thomas K Landauer, Peter W. Foltz, and Darrell Laham. An introduction to latent semantic analysis. *Discourse Processes*, 25(2-3):259–284, 1998.

[9] Hui Liu, Qingyu Yin, and William Yang Wang. Towards explainable NLP: A generative explanation framework for text classification. *CoRR*, abs/1811.00196, 2018.

[10] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[11] Tomas Mikolov, Quoc V. Le, and Ilya Sutskever. Exploiting similarities among languages for machine translation, 2013.

[12] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013.

[13] Ted Pedersen, Siddharth Patwardhan, and Jason Michelizzi. Wordnet::similarity - measuring the relatedness of concepts. 04 2004.

[14] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[15] Slav Petrov, Dipanjan Das, and Ryan T. McDonald. A universal part-of-speech tagset. *CoRR*, abs/1104.2086, 2011.

[16] Luis F Restrepo B and Julián González L. De Pearson a Spearman. *Revista Colombiana de Ciencias Pecuarias*, 20:183 – 192, 06 2007.

[17] Magnus Sahlgren. The distributional hypothesis. *The Italian Journal of Linguistics*, 20:33–54, 2008.

[18] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using T-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 11 2008.